Computación y programación en R: Tema 5

David V. Conesa Guillén

GEeitEma

Valencia Bayesian Research Group

Dept. d'Estadística i Investigació Operativa

Universitat de València

Tema 5: Programación de funciones y subrutinas

En este tema:

1.- Introducción: funciones y subrutinas en R. Expresiones agrupadas: sentencias entre llaves. Interacción con el bloc de notas.

2.- Órdenes para la ejecución condicional: `if` y `else`.

3.- Órdenes para la ejecución repetitiva en bucles y ciclos: `for`, `repeat` y `while`.

4.- Funciones: sintaxis y llamada.

5.- Nombres de argumentos y valores por defecto.

6.- El argumento "..."

7.- Funciones de control y parada: `warning`, `missing` y `stop`.

Tema 5: Programación de funciones y subrutinas

También:

8.- Debugging en R.

9.- Estrategias para mejorar el uso de R.

10.- Asignaciones dentro de las funciones.

11.- Ámbito o alcance de objetos.

12.- Personalización del entorno.

13.- Introducción a las clases y a la creación de librerías.

1.- Funciones y subrutinas en R. Expresiones agrupadas: sentencias entre llaves. Interacción con el bloc de notas.

- R es un lenguaje interactivo que nos permite crear objetos y analizarlos. Pero claramente R va mucho más allá.
- R es un lenguaje en constante evolución: permite ir creando nuevas estructuras que resuelven nuevos problemas que van apareciendo.
- R es un lenguaje de expresiones: *todos* los comandos ejecutados son funciones o expresiones que producen un resultado. Incluso las asignaciones son expresiones cuyo resultado es el valor asignado.
- Podemos crear explícitamente un objeto tipo expresión, con la función `expression()` y evaluarla con la función `eval()`:

Ejemplo

```
exp1<-expression(3+4)
exp2<-expression(sum(1:10))
exp3<-expression(b<-1:10)
eval(exp1); eval(exp2); eval(exp3)
```

Subrutinas: sentencias entre llaves.

- En general, una *subrutina* es un segmento de código que se escribe sólo una vez pero puede invocarse o ejecutarse muchas veces.
- Existen dos tipos:
  - *Procedimiento*: un grupo de expresiones entre llaves que no produce ningún resultado
  - *Función*: cuando dicho conjunto de código si produce un resultado
- Cuando agrupamos comandos o expresiones entre llaves `{expre.1; expre.2; ...; expre.m}`, las expresiones pueden ir:
  - separadas por ; en la misma línea o
  - separadas por cambio de línea
- En ambos casos el valor resultante de la subrutina es el resultado de la última expresión evaluada.
- Puesto que una subrutina es también una expresión, podemos incluirla entre paréntesis y usarla como parte de otra expresión.
- La mejor manera de poder ejecutar expresiones de varias líneas es a través de ficheros `script`.

Funciones y expresiones.

- La gran utilidad de las expresiones es que nos permiten ejecutar varios comandos de una única vez.
- Pero donde gana mayor utilidad esta forma de trabajar es a la hora de crear nuevos objetos que ejecuten diversas expresiones utilizando como entrada unos objetos (*argumentos*) y devolviendo otros objetos.
- Estos objetos (cuyo modo es `function`) constituyen las nuevas funciones de `R`, que se pueden utilizar a su vez en expresiones posteriores.
- En este proceso, el lenguaje gana en potencia, comodidad y elegancia.
- Muchas funciones del lenguaje `R` están escritas en código interno, otras utilizan conexiones a `C`, `Fortran`, etc. Pero otras muchas, como `mean` o `var`, están de hecho escritas en `R` y, por tanto, no difieren materialmente de las funciones que nosotros podamos escribir.
- Aprender a escribir funciones que nos puedan ser de futura utilidad es una de las mejores formas de conseguir que el uso de `R` nos sea cómodo y productivo.

2.- Ejecución condicional: `if` y `else`.

El lenguaje `R` tiene la posibilidad de ejecutar expresiones condicionalmente:

`if (expre1) expre2 else expre3`

Si `expre1=TRUE` calcula `expre2`
Si `expre1=FALSE` calcula `expre3`

Ejemplo

```
if (10>3) cat("SI 10>3 \n") else cat("NO 10>3 \n")
```

En la primera expresión podemos incluir varios requerimientos utilizando los operadores lógicos presentados en el tema 2.

Ejemplo

```
x<-0
if (is.numeric(x)&min(x)>0) rax<-sqrt(x) else stop("x debe
ser numérico y positivo \n")
```


Más órdenes para la ejecución condicional

El lenguaje `R` dispone de una versión vectorizada de `if`, que es `ifelse()`:

`ifelse( vec.test, vec.si.true, vec.si.false)`

Devuelve un vector cuya longitud es la del más largo de sus argumentos y cuyo elemento `i` es `vec.si.true[i]` si `vec.test[i]` es cierta, y `vec.si.false[i]` en caso contrario.

Ejemplo

```
y<- -5:5
y.logy<-ifelse(y>0,y*log(y),0)
round(y.logy,3)
```

Más órdenes para la ejecución condicional

Para evitar la concatenación de muchos `if`'s cuando tenemos varias posibilidades de ejecución, `R` dispone de la función `switch()`.

`switch(test, expre1,expre2,...,expren)`

Si test es un número `i`, calcula la expresión `i`.

`switch(test, nombre1=expre1,nombre2=expre2,...,nombren=expren)`

Si test es alfanumérico, calcula la expresión con dicho nombre

Ejemplo

```
switch(2,"1","2","3")
for(i in c(-1:3,9)) print(switch(i, 1,2,3,4))
switch("tres",uno="1",dos="2",tres="3")
switch("cuatro",uno="1",dos="2",tres="3",cat("Te has equivocado \n"))
x<-0 ; switch(x,1,2,3)
```


3.- Órdenes para la ejecución repetitiva en bucles y ciclos: `for`, `repeat` y `while`.

- Las órdenes de control para la ejecución repetitiva en bucles y ciclos deben utilizarse lo menos posible ya que ralentizan mucho los cálculos.
- Por la forma en la que `R` está diseñado, orientado a objetos, siempre que se pueda (casi siempre) utilizar las funciones vectorizadas `apply()`, `tapply()`, `sapply()` que veremos más adelante.
- Hay ocasiones en las que no hay más remedio que utilizar bucles y ciclos. `R` dispone de las herramientas necesarias para ello:
  - `for`
  - `while`
  - `repeat`

Ejecución repetitiva: `for` *y* `while`*.*

`for (name in values) expre`

La `expre` es evaluada asignado a `name` sucesivamente cada uno de los elementos de `values`.
Recordar que si la `expre` tiene más de un comando va entre llaves.

Ejemplo

```
for (i in 1:5) cat("caso ",i,"\n")
```

`while (condi) expre`

La `expre` es evaluada mientras la `condi` sea cierta.
Recordar que si la `expre` tiene más de un comando va entre llaves.

Ejemplo

```
i<-5
while (i >0) {cat("caso ",i,"\n"); i<-i-1}
```


Ejecución repetitiva: `repeat`*.*

`repeat expre`

La `expre` es evaluada de forma repetitiva.

La única forma de salir de un ciclo es con:

- `break` termina cualquier ciclo `for`, `while` o `repeat`.
- `next` dentro de `for`, `while` o `repeat` fuerza el comienzo de una nueva iteración.

Ejemplo

```
repeat {nlot<-sample(1:10,1,rep=T); if(nlot==5) break() else
cat("No es 5 es ",nlot,"\n")}
```

4.- Funciones: sintaxis y llamada.

- Hemos visto y utilizado de forma informal muchas funciones de R.
- Es posible ver su contenido tecleando únicamente su nombre.
- Una función se define con una asignación de la siguiente manera:

Sintaxis de una función

```
nombre <- function(arg1,arg2,...){expre}
```

- Cuidado con la asignación de nombres, evitar posible duplicidad. TinnR.
- `expre` es una expresión o grupo de expresiones (entre llaves). Recordar que el valor de la expresión será la última expresión del grupo evaluada.
- Para la evaluación de `expre` utilizamos argumentos `arg1,arg2,...`.
- El valor de `expre` es el valor que proporciona R en su salida y éste puede ser un simple número, un vector, una gráfica, una lista o un mensaje.

Llamada de una función

```
nombre(expre1,expre2,...)
```


Ejemplo

```
# Función que suma dos números
sumanumeros<-function(x,y) (x+y)
sumanumeros(3,5)

# ¿Y si quiero sumar vectores? La función también es válida
sumanumeros(c(2,3),c(3,5))
# ¿Y si los vectores no son del mismo tamaño?
sumanumeros(c(2,3),c(3,5,8))
# Veremos cómo añadir órdenes de control que paren la función.

# Función que calcula el factorial de un número natural
factorial<-function(x) prod(1:x)
factorial(8)

# Función que compara dos vectores, devuelve vector con los valores más grandes
grande<-function(x,y){
y.g<-y>x
x[y.g]<-y[y.g]
x
}
# la última sentencia es necesaria. ¿Qué ocurre si no la ponemos?
grande(1:5,c(1,6,2,7,3))
```

Ejemplo

```
# Función para comparar dos muestras independientes: test t
DosMuestras <- function(y1, y2) {
n1 <- length(y1); n2 <- length(y2)
yb1 <- mean(y1); yb2 <- mean(y2)
s1 <- var(y1); s2 <- var(y2)
s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
tst
}
# Para comparar la altura de las personas de esta clase por sexo:
datos.hombre <- scan()
datos.mujer <- scan()
result <- DosMuestras(datos.hombre, datos.mujer); result
```


5.- Nombres de argumentos y valores por defecto.

- Los argumentos deben darse en el orden en el que se han definido en la función.
- Sin embargo, cuando los argumentos se dan por nombre, "nombre.arg=objeto", el orden de los mismos es irrelevante.
- Además el nombre puede reducirse siempre que siga siendo distingui[i]le del resto de los argumentos.
- `args(nombre.funcion)` nos muestra los argumentos de cualquier función.

Ejemplo

```
# Son equivalentes:
grande(1:5, c(1,6,2,7,3))
grande(x=1:5, y=c(1,6,2,7,3))
grande(y=c(1,6,2,7,3), x=1:5)
grande(1:5, y=c(1,6,2,7,3))
```

Argumentos por defecto.

- En muchos casos hay funciones que tienen argumentos que deseamos tengan un valor por defecto.
- Si éste es omitido en la llamada, tomará el valor de la definición.
- La forma de hacerlo es incluir en la definición de la función: "nombre.argumento=valor.por.defecto"

Ejemplo

```
# Por defecto comparar con 0
grande<-function(x,y=0*x){
#si ponemos y=0 lo toma como número y no funciona
y.g<-y>x
x[y.g]<-y[y.g]
x
}
grande(c(-12:3))
grande(c(1,2),2:3)
```


6.- El argumento "..."

- En muchos casos nos interesa utilizar argumentos de otras funciones.
- El argumento "..." (tres puntos) nos permite incluir en la definición de nuestras funciones la posibilidad de llamar a nuestra función con otros argumentos que utilizarán funciones que estén dentro de la definición de nuestra función.

Ejemplo

```
# Función que calcula la media de cualquier número de vectores
media.total<-function(...) { mean(c(...)) }
media.total(1:4,-pi:pi)

# Función ejemplo del argumento ...
ejem.fun <- function(x, y, label = "la x", ...){
plot(x, y, xlab = label, ...) }
ejem.fun(1:5, 1:5)
ejem.fun(1:5, 1:5, col = "red")
```

7.- Funciones de control y parada: `warning`*,* `missing` *y* `stop`*.*

- En los ejemplos anteriores no hemos tenido ningún cuidado en comprobar si los argumentos son los apropiados, lo que podría habernos llevado a errores de sistema.
- `R` nos permite utilizar funciones para controlar y parar el funcionamiento de una función.
- Si hacemos comprobaciones y detectamos un error que no es grave, podemos llamar a la función `warning("mensaje")` que nos muestra el `mensaje` y la ejecución de la función continua.
- Si utilizamos la función `stop("mensaje")`, nos muestra el mensaje de error y deja de evaluar la función.
- La función `missing(argumento)` indica de forma lógica si un argumento no ha sido especificado.

Ejemplo

```
# Función media de varios vectores arreglada para parar
media.total<-function(...){
for (x in list(...)){
if (!is.numeric(x)) stop("No son numeros")
}
mean(c(...))
}
media.total("a",3)

# Función grande arreglada con avisos
grande<-function(x,y=0*x){
if (missing(y)) warning("Estamos comparando con 0")
y.g<-y>x
x[y.g]<-y[y.g]
x
}
grande(-3:3)
```

8.- Debugging en R.

- Cuando se produce una salida fuera de lo que esperábamos en una función que hemos programado, lo mejor es investigar el estado de las variables donde y cuando ha ocurrido el error.
- Este proceso se conoce en inglés como *debug*, de donde proviene la palabra *debugging*, que es la que se suele utilizar para indicar que se está procediendo a la búsqueda de errores en la programación.
- Existen varias posibilidades:
  - `traceback()` nos informa de la secuencia de llamadas antes del "crash" de nuestra función. Es muy útil cuando se producen mensajes de error incomprensibles.
  - Con la función `browser` podemos interrumpir la ejecución a partir de ese punto, lo que nos permite seguir la ejecución o examinar el entorno. Con "n" vamos paso a paso, con cualquier otra tecla se sigue la ejecución normal. "Q" para salir.
  - debug es como poner un `broswer` al principio de la función, con lo que conseguimos ejecutar la función paso a paso. Se sale con "Q".

Ejemplo

```
# Función mejorada para comparar dos muestras independientes: test t
ttest <- function(y1, y2, test="dos-colas", alpha=0.05) {
n1 <- length(y1); n2 <- length(y2) ; ngl<-n1+n2-2
s2 <- ((n1-1)*var(y1) + (n2-1)*var(y2))/ngl
tstat <- (mean(y1) - mean(y2))/sqrt(s2*(1/n1 + 1/n2))
area.cola <- switch(test,
"dos-colas" = 2 * (1 - pt(abs(tstat),ngl)),
inferior = pt(tstat,ngl),
superior = 1 - pt(tstat,ngl),
stop("el test debe ser dos-colas, inferior o superior" )
)
list(tstat=tstat, gl=ngl,
rechazar=if(!is.null(area.cola)) area.cola<alpha,
area.cola=area.cola)
}
# Esta función tiene errores, veamos cómo detectarlos
x1<-round(rnorm(10),1); x2<-round(rnorm(10)+1,1)
ttest(x1,x2); unlist(ttest(x1,x2)); t.test(x1,x2)
debug(ttest); ttest(x1,x2)
# pulsando n y después intro vamos viendo los pasos que va ejecutando
# y podemos ver que valores van tomando las variables
undebug(ttest)
```

GEeitEm

Ejemplo

```
ttest(x1,x2,inferior) # da un error, veamos que ha pasado
debug(ttest)
ttest(x1,x2,inferior)
# vemos donde ha ocurrido
traceback()
# vemos donde ha ocurrido
undebug(ttest)
ttest(x1,x2,"inferior")

# añadir un browser() en medio de la función, donde queramos
ttest(x1,x2,"inferior")
```


9.- Estrategias para mejorar el uso de R.

- Muchos usuarios de otros programas (sobre todo programadores) que llegan a R suelen tardar en sacarle provecho al uso de los cálculos vectorizados.
- La ventaja de los cálculos vectorizados es que operen sobre un vector completo en vez de sobre cada una de las componentes secuencialmente.
- El uso secuencial lleva a bucles innecesarios que sólo consiguen ralentizar la ejecución de una función.
- La mayoría de las veces que pensamos en utilizar un bucle es posible encontrar una forma alternativa que lo evita:
  - bien utilizando operaciones con vectores,
  - bien aplicando funciones de la familia `apply` a las diferentes componentes del objeto.

Evitando bucles.

Ejemplo

Suponer que queremos calcular el estadístico ji-cuadrado de Pearson para contrastar la hipótesis de independencia:

$$\chi_s^2 = \sum_{i=1}^{r}\sum_{j=1}^{s}\frac{(O_{ij}-E_{ij})^2}{E_{ij}}$$

donde $E_{ij} = \frac{O_{i.}O_{.j}}{O_{..}}$ son las frecuencias esperadas.
En principio podría parecer que para calcular el estadístico se necesitan dos bucles. ¿Seguro?

```
f <- matrix(c(23,45,43,22),ncol=2)
fi. <- f%*% rep(1,ncol(f))
f.j <- rep(1,nrow(f)) %*% f
e <- (fi. %*% f.j)/sum(fi.)
X2s <- sum((f-e)^2/e)
```


Variantes de la función apply

- `apply(x,margin,FUN,...)`: Aplica la función `FUN` a la dimensión especificada, donde `margin` 1 indica filas y 2 indica columnas.
- Para aplicar una función a todas las componentes de una lista tenemos la función `lapply(x, fun, ...)`
- Para aplicar una función a todas las componentes de una lista y simplificar el resultado en una estructura de vector, tenemos la función `sapply(x, fun, ..., simplify = TRUE)`.
- Para aplicar una función a un conjunto de valores seleccionados por medio de la combinación única de niveles de unos determinados factores utilizamos la función `tapply(x, INDEX, fun = NULL, ..., simplify = TRUE)` donde:
  - `x` es un objeto tipo atómico (habitualmente un vector)
  - `INDEX` es una lista de factores, todos de longitud la de `x`
  - `fun` es la función a aplicar (si es un + o un por, debemos entrecomillarlas)
  - `simplify`, si es FALSE, devuelve un array en modo lista.

Ejemplo

```
ejemplolista <- list(nombre="Pedro", casado=T,
       esposa="Mari a",no.hijos=3, edad.hijos=c(4,7,9))
lapply(ejemplolista,length); sapply(ejemplolista,length)

n <- 23; fac <- factor(rep(1:3, length = n), levels = 1:3)
table(fac)
tapply(1:n, fac, sum)
tapply(1:n, fac, sum, simplify = FALSE)
tapply(1:n, fac, range)
tapply(1:n, fac, quantile)

data(PlantGrowth)
attach(PlantGrowth)
tapply(weight,group,sum)
tapply(weight,group,mean)
tapply(weight,group,log)
tapply(weight,group,function(x)sum(log(x)))
```


10.- Asignaciones dentro de las funciones.

- Recordar que una función se define con una asignación de la siguiente manera:

Sintaxis de una función

```
nombre <- function(arg1,arg2,...){expre}
```

- Cualquier asignación ordinaria realizada dentro de una función es local y temporal y se pierde tras salir de la función.

Ejemplo

```
sumav<-function(x,y){ x <- x+y ; x }
```

- Por tanto, la asignación `x <- x+y` no afecta al valor del argumento de la función en que se utiliza.
- Como veremos en la siguiente sección, a veces es conveniente realizar asignaciones globales y permanentes dentro de una función. Para ello, utilizaremos el operador de "superasignación", `<<-`, o la función `assign`. Ver la ayuda para una explicación más detallada.

11.- Ámbito o alcance de objetos.

- Los símbolos que ocurren en el cuerpo de una función pueden dividirse en tres:
  - *parámetros formales*, los argumentos de la función, su valor es lo que pasamos entre paréntesis.
  - *variables locales*, las que se asignan dentro de la función. Su nombre puede ser cualquiera y toman el valor asignado en ese momento sólo dentro de la función.
  - *variables libres*, las no incluidas en los apartados anteriores. Su valor se busca de forma secuencial en los entornos que han llamado a la función.
- El ámbito o el alcance son reglas utilizadas por el "evaluador" para encontrar el valor de uno de los símbolos anteriores.
- Cada lenguaje de programación tiene un conjunto de esas reglas.
- En R las reglas son simples, aunque existen mecanismos para cambiarlas.

Ámbito o alcance de objetos.

- El mecanismo que utiliza R se denomina *lexical scoping*: "las características de una variable en el momento en el que la expresión se crea se utilizan para asignar valores a cualquier símbolo de la expresión".
- En otras palabras: en R la asignación de valor a una variable libre se realiza consultando el entorno en el que la función se ha creado, *ámbito léxico*.
- Es diferente del que utilizan otros programas como S, que es *static scoping*.
- Para más información sobre el tema ver:
  - *Frames, environments and scope in R and S-PLUS* de J. Fox en `http://cran.r-project.org/doc/contrib/Fox-Companion/appendix.html` o
  - las secciones 3.5 y 4.3.4 del manual *The R language definition*.

Ámbito o alcance de objetos.

Ejemplo

```
# Definimos una función al principio de una sesión de R
verfun<-function(x) {
y<-2*x
print(x) # x en un parámetro formal.
print(y) # y es una variable local.
print(z) # z es una variable libre.
}
# Llamamos a la función
verfun(8)
# [1] 8
# [1] 16
# Error: Object "z" not found
# Busca la z en el entorno que la ha creado y no la
encuentra.
```


Ámbito o alcance de objetos.

Ejemplo

```
# Continuamos con el ejemplo anterior.
# Definimos la variable z.
z<-3
# Llamamos a la función
verfun(8)
# [1] 8
# [1] 16
# [1] 3
# ahora si encuentra z y nos da su valor.
```

Ámbito o alcance de objetos.

Ejemplo

```
# Definimos una nueva función ver2fun
ver2fun<-function(x){
z<-10
cat("z dentro ver2fun vale",z,"\n")
verfun(z)
}
# Llamamos a la función
ver2fun(6)
# z dentro ver2fun vale 10
# [1] 10
# [1] 20
# [1] 3
z
# [1] 3
# El valor de z del primer entorno que creó a la función
```


Uso de `missing`.

Para comprobar si un argumento de una función tiene valor podemos utilizar la función `missing(x)`. Esta función sólo es fiable si `x` no ha sido alterado desde que ha entrado en la función.

Ejemplo

Esta función muestra como podemos realizar una gráfica dependiendo de los argumentos que tengamos: con un par de vectores gráfica de uno vs. el otro, con un único vector, gráfica del vector vs. sus índices.

```
myplot <- function(x,y) {
if(missing(y)) {
y <- x
x <- 1:length(y)
}
plot(x,y)
}
```

Ámbito o alcance de objetos.

- Las funciones pueden ser recursivas e, incluso, pueden definir funciones en su interior.
- Pero cuidado, dichas funciones, y por supuesto las variables, no son heredadas por funciones llamadas en marcos de evaluación superior, como lo serían si estuviesen en la trayectoria de búsqueda.
- Además, dentro del ámbito lexicográfico es posible para conceder a las funciones un estado cambiante.
- El operador asignación especial «- comprueba los entornos creados desde el actual hasta el primero hasta encontrar uno que contenga el símbolo al que estamos asignando y cuando lo encuentra, sustituye su valor en dicho entorno por el valor de la derecha de la expresión. Si se alcanza el nivel superior, correspondiente al entorno global, sin encontrar dicho símbolo, entonces lo crea en él y realiza la asignación.

Ámbito o alcance de objetos.

- Veamos cómo utilizar `R` para representar el funcionamiento de una cuenta bancaria. Una cuenta bancaria necesita tener un balance o total, una función para realizar depósitos, otra para retirar fondos y una última para conocer el balance.
- Crearemos pues tres funciones dentro de `anota.importe`, devolviendo una lista que las contiene. Al ejecutar `anota.importe` toma un argumento numérico, `total` y devuelve una lista que contiene las tres funciones.
- Puesto que estas funciones están definidas dentro de un entorno que contiene a `total`, éstas tendrían acceso a su valor. Utilizamos el operador de asignación especial, «-, para cambiar el valor asociado con `total`.

Ejemplo

```
# anota.importe <- function(total) {
list(
deposito = function(importe)
{ if(importe <= 0) stop("Los depósitos deben ser positivos!\n")
total «- total + importe
cat("Depositado",importe,". El total es", total, "\n \n") },
retirada = function(importe)
{ if(importe > total) stop("No tiene tanto dinero!\n")
total «- total - importe
cat("Descontado", importe,". El total es", total,"\n \n") },
balance = function() { cat("El total es", total,"\n \n") }
)
}

Emili <- anota.importe(100); Emili
Roberto <- anota.importe(200)
Emili$retirada(30); Emili$balance()
Roberto$balance()
Emili$deposit(50); Emili$balance()
Emili$retirada(500)
```


12.- Personalización del entorno.

- Es posible adoptar el entorno de trabajo de R a nuestras necesidades de varias formas.
  - Existe un archivo de inicialización del sistema denominado Rprofile. Sus órdenes se ejecutan cada vez que se comienza una sesión de R, sea cual sea el usuario.
  - También es posible disponer en cada directorio de un archivo de inicialización especial propio denominado .Rprofile (en principio archivo oculto). Permite controlar el espacio de trabajo y disponer de diferentes métodos de inicio para diferentes directorios de trabajo.
  - Además se puede usar las funciones especiales .First y .Last. Si existe la función .First() (en cualquiera de los dos archivos anteriores o en el archivo de imagen .RData) se ejecutará al comienzo de la sesión de R, y por tanto puede utilizarse para inicializar el entorno.
- En resumen, la secuencia en que se ejecutan los archivos es, Rprofile, .Rprofile, .RData y por último la función .First().

Ejecución no interactiva.

- Existen dos formas de ejecutar comandos de manera no interactiva:
  - Con `source`: abrimos una sesión de `R` y ejecutamos `source("micodigo.R")`
  - Con `BATCH`: `Rcmd BATCH micodigo.R`
- `source` es en ocasiones más útil porque informa inmediatamente de errores en el código.
- `BATCH` no informa, pero no requiere tener abierta una sesión (se puede correr en el background).
- La función `source` nos permite hacer explícitos los comandos con la opción `source(my.file.R, echo = TRUE)`.
- La función `sink()` es la inversa de `source` (lo manda todo a un fichero).

13.- Introducción a las clases y a la creación de librerías.

- `R` es básicamente un lenguaje orientado a objetos.
- La idea central de un lenguaje de este tipo se basa en dos conceptos:
  - Una *clase* es una definición de un objeto. Una clase contiene varios "slots" que sirven para contener información específica de la clase. Todo objeto del lenguaje debe pertenecer a una clase.
  - Los cálculos ser realizan a través de los *métodos*. Son funciones especializadas que permiten realizar cálculos específicos sobre los objetos, habitualmente de una clase específica (de aquí lo de la orientación a objetos).
- En `R`, las funciones genéricas se utilizan para determinar el método apropiado (basta teclear cualquier función, p.e. `mean`).
- Para más información sobre el tema ver la sección 5 del manual *The R language definition*.
- Se pueden crear paquetes, con nuestras funciones, que se comporten igual que los demás paquetes. Ver el manual *Writing R extensions*.

Licencia de este material

Más info: http://creativecommons.org/licenses/by-sa/3.0/es/

Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

Compartir bajo la misma licencia. Si transforma o modifica esta obra para crear una obra derivada, sólo puede distribuir la obra resultante bajo la misma licencia, una similar o una compatible.